REPORT DOCUMENTATION PAGE	Form Approved OMB NO. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggesstions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA, 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any oenalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 22-09-2009	2. REPORT TYPE Final Report	3. DATES COVERED (From - To) 15-Sep-2006 - 14-Sep-2009

4. TITLE AND SUBTITLE Real-Time Computing on Multicore Platforms	5a. CONTRACT NUMBER W911NF-06-1-0425
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 611102
6. AUTHORS James H. Anderson	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAMES AND ADDRESSES University of North Carolina - Chapel Hill Office of Sponsored Research 104 Airport Drive, Suite 2200, CB 1350 Chapel Hill, NC 27599 -1350	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211	10. SPONSOR/MONITOR'S ACRONYM(S) ARO 11. SPONSOR/MONITOR'S REPORT NUMBER(S) 49365-CS.55

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; Distribution Unlimited

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT
The objective of this project is to develop real-time scheduling and synchronization algorithms that are well-suited for multicore platforms and to implement these algorithms within a real OS. Towards this objective, new multiprocessor real-time scheduling algorithms have been developed that are optimized to deal with specific cache layouts and performance asymmetries in

15. SUBJECT TERMS
real-time, multicore, scheduling, synchronization, linux

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James Anderson
U	U	U	SAR		19b. TELEPHONE NUMBER 919-962-1757

Standard Form 298 (Rev 8/98)
Prescribed by ANSI Std. Z39.18

Real-Time Computing on Multicore Platforms

Final Report

James H. Anderson

September 2009

1 Statement of the Problem Studied

Multicore architectures, which contain multiple processing cores on a single chip, have been adopted by most chip manufacturers due to the thermal- and power-related limitations of single-core designs. Most chip manufacturers have released dual-core chips, Intel and AMD each have four-core chips on the market, and Sun's Niagara and more recent Niagara 2 processors are eight-core chips with multiple hardware threads per core. Furthermore, Intel has announced plans to release chips with as many as 80 cores within five years [17].

In most proposed multicore platforms, different cores share on-chip caches. To effectively exploit the available parallelism in these systems, such caches must not become performance bottlenecks. In fact, the issue of efficient cache usage on multicore platforms is one of the most important problems with which chip makers are currently grappling. In this project, we sought to address this issue in the context of soft real-time applications.

In our work, we mostly (but not exclusively) considered real-time systems that are defined as a collection of periodic tasks. A *periodic task* is invoked repeatedly, and each such invocation is called a *job* of the task. In the variant of the periodic model that we mostly focused on, each task T is characterized by a *period* $T.p$ and a per-job *execution cost* $T.e$: every $T.p$ time units, a new job of T is released that executes for $T.e$ time units. The quantity $T.e/T.p$ denotes the *utilization* of T. Each job is assigned a *deadline* corresponding to the release time of the next job of the same task. In a *hard* real-time system, job deadlines should never be missed. However, in a *soft* real-time system, deadline misses can sometimes be tolerated. Such misses are constrained to be within a specified per-task *tardiness threshold*: a job with a deadline at time d will be guaranteed to complete execution no later than time $d + \delta$, where δ is the tardiness threshold of the corresponding task. Such a guarantee ensures that each task T receives a processor share close to its utilization, but allows some leeway in scheduling. A real-time task system is called *schedulable* if all of its timing constraints (hard and soft) can be guaranteed. *Schedulability tests* are used to determine if such guarantees can be made.

In the context of multicore platforms, care must be taken when deciding which jobs should be *co-scheduled* to execute at the same time. In particular, some co-scheduling choices might be *constructive* while others are *destructive*. Constructive choices, such as co-scheduling jobs that share data, decrease shared-cache miss rates. On the other hand, destructive choices may increase shared-cache miss rates. This might happen, for example, if a set of jobs is co-scheduled that together have a combined working-set size that exceeds the capacity of the last level of shared cache.

In light of these observations, we sought in this project to solve the problem of devising multiprocessor scheduling policies that

> *schedule tasks so that each job of every task completes within its allowed tardiness bound, and shared-cache miss rates are kept reasonably low by encouraging constructive co-scheduling choices and discouraging destructive co-scheduling choices.*

Additionally, we sought to implement and evaluate such polices within a real operating system, and to also handle such complexities as task synchronization, dynamic task behaviors, and mixed real-time/non-real-time workloads.

2 Summary of the Most Important Results

Addressing the problem stated above led to many new results being obtained pertaining to real-time multiprocessor systems. This research has led to the publication of eight journal papers and 38 conference papers. The following paragraphs present an overview of our most most important results.

Algorithmic foundations. On multiprocessors, two basic approaches exist for scheduling real-time systems: *partitioning* and *global scheduling*. Under partitioning, tasks are statically assigned to processors and those assigned to each processor are scheduled upon it using a uniprocessor scheduling algorithm. Under global scheduling, tasks are scheduled from a single run-queue and may migrate among processors. As part of this project, we conducted the first ever comparisons of such approaches based upon real implementations [8, 15]. Our studies showed that, for hard real-time systems, partitioning algorithms are usually preferable, while for soft real-time systems, global algorithms are better. This is due to the following reasons. In the hard real-time case, most partitioning and global-scheduling approaches have rather similar schedulability tests in the absence of overheads. As a result, partitioning approaches tend to be better because they have lower run-time overheads. (Techniques for assessing schedulability are impacted by such overheads.) In contrast, in the soft real-time case, partitioning approaches are subject to bin-packing limitations that can be eliminated through the use of global algorithms. In particular, we have shown that most global algorithms are capable of ensuring bounded deadline tardiness on an m-processor platform for any periodic task system with total utilization at most m (i.e., that does not over-utilize the platform) [16, 18]. This is true even if tasks are required to execute non-preemptively (in which case, system overheads are very low, because migration costs are low). This result was shown to apply to a wide class of global algorithms wherein each job's priority is defined by a point in time (e.g., a deadline)—such points are even allowed to vary dynamically at runtime. In contrast, there exist task systems with total utilization slightly higher than $m/2$ that no partitioning scheme can schedule, even if bounded deadline tardiness is allowed. Such limitations are the reason for the better performance of global algorithms (in terms of schedulability) in the soft real-time case.

Cache- and platform-aware real-time scheduling algorithms. We have developed several new global scheduling algorithms that take into account the hardware characteristics of a multicore platform to improve performance. Of most relevance to the problem stated in Section 1 is a cache-aware global scheduling algorithm that is the central result of John Calandrino's Ph.D. dissertation [10]. This algorithm exploits the tardiness result mentioned in the previous paragraph, which allows a job's priority to be varied at runtime without causing tardiness to become unbounded. In the cache-aware algorithm, if it is constructive to co-schedule a set of jobs, then this is encouraged by increasing the priority of all jobs in the set whenever any one of them is scheduled [11]. Encouraging desirable co-scheduling choices has the effect of also making undesirable choices less likely. To determine which choices are desirable, a cache profiler was incorporated into the scheduler [12]. This profiler estimates the cache footprint of each ready job by using hardware performance counters. This cache-aware scheduling algorithm was shown in several experimental studies to lessen cache miss rates in comparison to non-cache-aware algorithms.

In other work, we developed new scheduling algorithms for *asymmetric multicore platforms*, or AMPs [14]. In the AMP architecture that we considered, a mixture of "fast" and "slow" cores are present. Such platforms have been suggested as a means for dealing with workloads in which some tasks are inherently sequential (and thus may benefit from being assigned to a fast core) while others are parallelizable (and thus may execute across many slow cores). In other work, we have investigated very large multicore platforms with hierarchical cache layouts [13]. We found that, for such platforms, an approach that mixes aspects of partitioning and global scheduling is preferable. In particular, while task migrations within a cluster of cores that share some lower-level cache might be acceptable, migrations among cores that are "far apart" in the cache hierarchy are expensive.

Prototype development. The experimental studies described above were conducted using a Linux-based system developed by us called LITMUS$^{\mathrm{RT}}$ (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems) [1, 3, 7, 9, 15, 19]. LITMUS$^{\mathrm{RT}}$ extends Linux (currently, version 2.6.24) by allowing different (multiprocessor) scheduling algorithms to be linked as plug-in components. The development of LITMUS$^{\mathrm{RT}}$ has been a major component of this project. LITMUS$^{\mathrm{RT}}$ is currently running in our lab on four machines: (1) a four-socket, single-core-per-socket Intel

Xeon platform with 2.7 GHz cores; (2) an eight-core, single-socket Sun UltraSPARC T1 (codename "Niagara") with 1.2 GHz cores; (3) a four-core, single-socket Intel Xeon E5420 ("Core" architecture) with 2.5 GHz cores; and (4) a four-core, single-socket Intel Core i7 ("Nahelem" architecture) with 2.66 GHz cores. As a result of our LITMUS$^{\mathrm{RT}}$-related work, key Linux developers are now planning to incorporate "hooks" into the scheduling code of mainline Linux (as LITMUS$^{\mathrm{RT}}$ does) so that researchers can define and experiment with different scheduling policies. To the best of our knowledge, LITMUS$^{\mathrm{RT}}$ is the only (published) system wherein global multiprocessor real-time scheduling algorithms are implemented in a real operating system.

A scheduling framework for mixed real-time and non-real-time workloads. We have developed and implemented in LITMUS$^{\mathrm{RT}}$ a hierarchical scheduling framework that allows hard and soft real-time tasks and best-effort jobs to be simultaneously supported and be temporally isolated from one another [3]. In this framework, hard real-time tasks are statically assigned to processors (i.e., are partitioned) and are accorded higher priority than soft real-time tasks and best-effort jobs, both of which are globally scheduled. Capacity reclamation techniques are used to improve best-effort response times and soft real-time tardiness. Spare processing capacity arises that can be reclaimed when jobs of real-time tasks (soft or hard) execute for less than their reserved allocations. In experiments conducted using LITMUS$^{\mathrm{RT}}$, the usage of capacity reclamation resulted best-effort response times approaching that of an idle system.

A new real-time multiprocessor synchronization protocol. Lock-based synchronization can be problematic in real-time systems due to the potential of *priority inversions* existing. A priority inversion is said to exist when a higher-priority job is forced to wait on a lower-priority job to release a lock. If the resulting waiting time is excessive, then the higher-priority job may exceed its tardiness threshold. Real-time synchronization protocols seek to limit the durations of priority inversions so that such violations do not occur. Such protocols have been well-studied in the case of uniprocessor systems. However, much less work has been done in the multiprocessor case. Multiprocessor real-time synchronization protocols tend to be much more complicated than uniprocessor ones, because waiting dependencies can be more complex.

In prior work on multiprocessor real-time synchronization, global scheduling algorithms were not considered at all. To enable locking to be supported under such algorithms, we developed and implemented in LITMUS$^{\mathrm{RT}}$ a new real-time synchronization protocol called the *flexible multiprocessor locking protocol* (FMLP) [2, 5, 4, 9]. The FMLP provides support for both spin-based and semaphore-based locking under both global and partitioned scheduling. The FMLP is the only real-time synchronization scheme known to us that can be used in global scheduling algorithms. In addition, its impact on real-time schedulability in partitioned systems is much less than is the case with prior schemes. In recent work, we also extended the spin-based variant of the FMLP to support reader/writer locks [6]. Such locks allow read accesses to occur concurrently (write accesses are exclusive). This can greatly reduce blocking times in workloads wherein locks are used to protect read-mostly data.

References

[1] A. Block, B. Brandenburg, J. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time systems. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 23–33, July 2008.

[2] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80. IEEE, August 2007.

[3] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 61–70. IEEE, July 2007.

[4] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 105–124, December 2008.

[5] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, M-PCP, D-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\text{RT}}$. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 185–194, August 2008.

[6] B. Brandenburg and J. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 184–193, July 2009.

[7] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\text{RT}}$: A status report. In *Proceedings of the 9th Real-Time Workshop*, pages 107–123. Real-Time Linux Foundation, November 2007.

[8] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169. IEEE, December 2008.

[9] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE, April 2008.

[10] J. Calandrino. *On the Design and Implementation of a Cache-Aware Soft Real-Time Scheduler for Multicore Platforms*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2009.

[11] J. Calandrino and J. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 209–308, July 2008.

[12] J. Calandrino and J. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 194–204, July 2009.

[13] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–256. IEEE, July 2007.

[14] J. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 101–110. IEEE, April 2007.

[15] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\text{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123. IEEE, December 2006.

[16] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341. IEEE, December 2005.

[17] C. Farivar. Intel Developers Forum Roundup: Four cores now, 80 cores later. http://www.engadget.com/2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/, 2006.

[18] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 413–422. IEEE, 2007.

[19] UNC Real-Time Group. LITMUS$^{\text{RT}}$ project. http://www.cs.unc.edu/˜anderson/litmus-rt/.